

== 운영체제 최종보고서 ==

< 1 팀 >

과 목	운영체제
담당 교수	박재근
전공	전자정보공학부 IT 융합학과
이름	팀장: 우승민 정지현, 임지혁, 한성규

- 목 차 -

I. 서론	
II. 명령어 구현	
1. 구현 목록	
2. 동작 과정	
III. 채팅 프로그램 및 스케줄러.....	
1. IPC	
2. 구조	
3. Client 동작 과정	
4. Server 동작 과정	
III-1. Scheduling	
1. Round Robin	
2. 사용이유	
III-2. Multi Threading.....	
1. 이론적 배경	
2. 구현	
III-3. 동기화 처리.....	
1. 동기화	
2. 최종 채팅 프로그램	
IV. 결론.....	
V. 참고문헌.....	

I. 서론

본 프로젝트에서 우리 1 팀은 운영체제 전반적인 내용을 학습하고 구현할 수 있는 부분들을 선정해 구현하였다. C 언어 프로그래밍이 익숙하지 않은 팀원이 많기에 높은 난이도의 구현보다는 적절한 수준의 난이도를 선택하여 운영체제 개념에 대한 학습과 그에 대한 코드 구현 경험, 팀프로젝트에 대한 협업 경험을 가장 큰 목표로 본 프로젝트를 시작하였다.

1 팀에서 선정한 운영체제 구현 목록은 첫번째로 파일 관련 터미널 명령어들을 찾아보고 그에 대한 동작 방식을 이해하며 오픈소스를 활용해 ls, cp, mkdir 등 파일 관련 터미널 명령어들을 직접 구현해 보았다. 두 번째로는 동일한 시스템에서 실행 중인 여러 프로세스가 서로 데이터를 교환하거나 통신하기 위한 메커니즘인 IPC(Inter-Process Communication, 프로세스 간 통신)의 방법 중 소켓을 활용해 채팅 프로그램을 구현하였다. 클라이언트 서버 구조로 하나의 로컬 서버에 두 개 이상의 클라이언트가 접속해 소켓 통신으로 서로 통신을 하는 구조이다. 더 나아가 구현한 채팅 프로그램을 활용해 스케줄링을 통해 두 채팅 클라이언트의 동기화 문제를 해결하려고 하였으나 SIGSTOP 명령어 수행과정에서 실행 BASH 가 꺼지는 현상으로 인해 스케줄러의 구현은 성공하였으나 다른 실행 파일에서 직접 적용하기는 실패하였다. 이런 이유로 채팅 클라이언트의 동기화 문제를 Mutex 를 통해 해결하고자 하였고 다수의 채팅 클라이언트 접속시 생기는 문제 상황을 실험하고 그에 대한 해결책으로 Mutex 를 적용해 해결 하려는 시도를 하였다.

우리는 이 프로젝트를 통해 운영체제에 대한 내용을 이론이 아닌 코드적으로 이해하는 값진 시간을 보냈고 각자 시행 착오를 통해 실제 구현까지 해보며 C 언어 프로그래밍에 대한 실력을 키울 수 있었다. 이 과정에서 실제 현업에서 사용하는 오픈소스의 활용, 최근 개발자들 사이에서 많은 이슈가 되고있는 Chat GPT 에 대한 활용 능력, 코드의 구조 파악 및 코드 리딩(Reading) 능력, 많은 회사에서 요구하고 있는 Github 에 대한 활용 능력, 팀원들과 직접 아이디어를 고민하고 커뮤니케이션 하는 능력까지 얻을 수 있었다.

II. 명령어 구현

1. 구현 목록

cp	파일이나 디렉토리를 복사
mkdir	새로운 디렉토리를 생성
cat	파일 내용을 화면에 출력
find	파일 시스템에서 파일을 검색하는 데 사용
ls	디렉토리에 있는 파일의 목록을

	나열하는
pwd	현재 디렉터리의 전체 경로를 화면에 표시
mv	파일이나 디렉터리의 이름을 변경하거나 다른 디렉터리로 옮길 때 사용

2. 동작 과정 - 각 명령어들을 구현 내용 설명, 코드 해석.

1) cp

```
void writefile(const char *in_f, const char *out_f) {
    int in_o, out_o;
    int read_o;
    char buf[1024];

    in_o = open(in_f, O_RDONLY);
    if (in_o == -1) {
        perror("Error opening input file");
        return;
    }

    out_o = open(out_f, O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);
    if (out_o == -1) {
        perror("Error opening output file");
        close(in_o);
        return;
    }

    while ((read_o = read(in_o, buf, sizeof(buf))) > 0) {
        if (write(out_o, buf, read_o) != read_o) {
            perror("Error writing to output file");
            close(in_o);
            close(out_o);
            return;
        }
    }

    if (read_o == -1) {
        perror("Error reading from input file");
    }
}
```

이 함수는 지정된 입력 파일을 읽고 지정된 출력 파일에 쓴다.

open() 함수를 통해 입력받은 파일을 열고, **open()** 함수로 출력 파일 열고 만약 파일이 없으면 생성하고 내용을 덮어쓰기 모드로 열기를 진행한다.

다음으로 입력 파일에서 데이터를 읽은 뒤 출력 파일에 쓰기 작업을 진행한다.

```

void cpok(const char *source, const char *destination)
{
    if (access(destination, F_OK) == 0) {
        char conin = 'h';
        while (conin != 'y' && conin != 'n') {
            printf("대상 파일이 이미 존재합니다. 덮어쓰시겠습니까?(y/n)\n");
            conin = getchar();
        }

        if (conin == 'y') {
            writefile(source, destination);
            printf("복사 성공\n");
        }
    } else {
        writefile(source, destination);
        printf("복사 성공\n");
    }
}

```

이 함수는 소스 파일을 대상 파일로 복사한다. 대상 파일이 이미 존재하는 경우 사용자에게 덮어쓸지 여부를 묻는다. 대상 파일이 존재하는지 확인한 뒤 존재하지 않는다면 바로 복사를 진행한다. 만약 대상파일이 이미 존재한다면 덮어쓰기를 할건지를 사용자에게 출력한다. 위의 두 함수를 통해서 **cp** 명령어가 실행된다.

2) mkdir

```

int create_directoryok(const char* dirname) {
    int status = mkdir(dirname, 0700);
    printf("디렉토리 생성이 완료되었습니다. \n");
    return status;
}

```

mkdir() 함수는 C 언어 라이브러리에 있는 함수로 mkdir 명령어와 같은 기능을 함으로 위 함수를 사용해 명령어를 구현했다.

3) cat

```

#define BUF_SIZE 1024

char *g_name;

```

버퍼의 크기, 프로그램의 이름 저장을 위한 전역변수를 선언하였다.

```

void ft_putstr(char *str) {
    while (*str)
        write(1, str++, 1);
}

```

문자열을 출력하는 함수이다.

```

void print_error(char *file) {
    ft_putstr(basename(g_name));
    ft_putstr(": ");
    ft_putstr(file);
    ft_putstr(": ");
    ft_putstr(strerror(errno));
    ft_putstr("\n");
    errno = 0;
}

```

파일 관련 오류 메시지를 출력하는 함수이다.

```

char *ft_strcpy(char *str1, char *str2) {
    int i = 0;
    while (str2[i]) {
        str1[i] = str2[i];
        i++;
    }
    str1[i] = 0;
    return str1;
}

```

문자열을 복사하는 함수이다.

```

char *ft_strcat(char *str1, char *str2) {
    char *ptr = str1;
    while (*ptr) ++ptr; // 이 부분이 필요함
    while (*str2) {
        *ptr++ = *str2++;
    }
    *ptr = 0;
    return str1;
}

```

문자열을 이어 붙이는 함수이다.

```

void write_file(int fd, char *file, int number) {
    long size;
    unsigned char buf[BUF_SIZE];
    while ((size = read(fd, buf, BUF_SIZE)) > 0) {
        if (errno) {
            print_error(file);
            return;
        }
        if (number == 1)
            strcpy(file, (char *)buf);
        else if (number == 2)
            strcat(file, (char *)buf);
    }
}

```

파일 디스크립터로부터 읽은 내용을 다른 파일에 쓰는 함수이다.

```

void display_file(int fd, char *file) {
    long size;
    unsigned char buf[BUF_SIZE];
    while ((size = read(fd, buf, BUF_SIZE)) > 0) {
        if (errno) {
            print_error(file);
            return;
        }
        write(1, buf, size);
    }
}

```

파일 디스크립터로부터 읽은 내용을 표준 출력에 출력하는 함수이다.

```

int fd;
int i;
g_name = av[0];
if (ac == 1)
    display_file(0, 0);
else
{
    i = 0;
    while (++i < ac)
    {
        if (strcmp(av[i], ">") == 0)
        {
            fd = open(av[2], O_CREAT | O_WRONLY | O_TRUNC, 0644);
            write_file(fd, av[2], 1);
        }
        else if (strcmp(av[i], ">>") == 0)
        {
            fd = open(av[2], O_CREAT | O_WRONLY | O_APPEND, 0644);
            write_file(fd, av[2], 2);
        }
        else
        {
            if ((fd = open(av[i], O_RDONLY)) == -1)
            {
                print_error(av[i]);
                continue ;
            }
            display_file(fd, av[i]);
            close(fd);
        }
    }
}

```

위의 정의한 함수들을 사용해 **cat** 기능을 전체적으로 동작하는 코드의 전문이다.

cat 명령어에서 다음에 나오는 텍스트가 없다면 아무것도 출력하지 않고 뒤의 텍스트가 있다면 첫 번째 인자가 '>' 인 경우 새 파일에 쓰고, '>>' 인 경우 기존 파일에 내용을 추가하는 방식으로 동작한다.

그 외의 경우 파일을 읽어 표준 출력에 출력한다.

4) find

```

void findFile(const char *dirName, const char *fileName) {
    DIR *dir;
    struct dirent *entry;
    struct stat statbuf;

    if ((dir = opendir(dirName)) == NULL) {
        fprintf(stderr, "Cannot open directory: %s\n", dirName);
        return;
    }

    while ((entry = readdir(dir)) != NULL) {
        char path[1024];
        snprintf(path, sizeof(path), "%s/%s", dirName, entry->d_name);

        if (stat(path, &statbuf) == -1) {
            fprintf(stderr, "Cannot access the file: %s\n", path);
            continue;
        }

        if (S_ISDIR(statbuf.st_mode)) {
            if (strcmp(entry->d_name, ".") != 0 && strcmp(entry->d_name, "..") != 0) {
                findFile(path, fileName);
            }
        } else {
            if (strcmp(entry->d_name, fileName) == 0) {
                printf("Found: %s\n", path);
            }
        }
    }
    closedir(dir);
}

```


`opendir` 함수를 사용하여 디렉토리를 열고 `readdir` 함수를 사용하여 현재 열고 있는 디렉토리 내의 각 항목을 읽는다.

5) ls

```
void mode_to_str(mode_t mode, char str[11])
{
    strcpy(str, "-----");
    if(S_ISREG(mode))      str[0] = '-';
    else                  str[0] = 'd';

    //user
    if(mode & S_IRUSR) str[1] = 'r';
    if(mode & S_IWUSR) str[2] = 'w';
    if(mode & S_IXUSR) str[3] = 'x';
    //group
    if(mode & S_IRGRP) str[4] = 'r';
    if(mode & S_IWGRP) str[5] = 'w';
    if(mode & S_IXGRP) str[6] = 'x';
    //other
    if(mode & S_IROTH) str[7] = 'r';
    if(mode & S_IWOTH) str[8] = 'w';
    if(mode & S_IXOTH) str[9] = 'x';
}
```

`mode_to_str` 함수는 파일 모드(권한)를 문자열로 변환 한다. 기본 문자열을 초기화하고 파일 모드에 따라 적절한 문자를 설정한다.

```
char *uid_to_name(uid_t uid)
{
    struct passwd *ppwd;
    static char number[10];
    ppwd = getpwuid(uid);
    if(ppwd == NULL){
        sprintf(number, "%d ", uid);
        return number;
    }
    return ppwd->pw_name;
}
```

`uid_to_name` 함수는 사용자 ID 를 사용자 이름으로 변환한다. `getpwuid(uid)`는 사용자 ID 에 대한 `passwd` 구조체를 얻는다. 사용자가 없으면 사용자 ID 를 문자열로 반환하고 있으면 사용자 이름을 반환한다.

```

char *gid_to_name(gid_t gid)
{
    struct group *pgroup;
    static char number[10];
    pgroup = getgrgid(gid);
    if(pgroup == NULL){
        sprintf(number, "%d ", gid);
        return number;
    }
    return pgroup->gr_name;
}

```

gid_to_name 함수는 그룹 ID 를 그룹 이름으로 변환한다. getgrgid(gid)는 그룹 ID 에 대한 group 구조체를 얻는다. 그룹이 없으면 그룹 ID 를 문자열로 반환하고 있으면 그룹 이름을 반환한다.

```

void show_info(const char *dir, const char *filename)
{
    struct stat s;
    char str_mode[11];

    int length_dir = strlen(dir);
    int length_file = strlen(filename);
    char full_file_name[length_dir + length_file + 2];
    strcpy(full_file_name, dir);
    full_file_name[length_dir] = '/';
    strcpy(full_file_name + length_dir + 1, filename);

    int ret = stat(full_file_name, &s);
    if(ret != 0){
        perror("read stat error");
        exit(1);
    }
    mode_to_str(s.st_mode, str_mode);
    printf("%s ", filename);
}

```

show_info 함수는 파일에 대한 정보를 출력한다. stat 함수를 사용하여 파일 정보를 얻는다. 파일 모드를 문자열로 변환하고 파일 이름을 출력한다.

```

int compare(_Node *p, _Node *q)
{
    return strcmp(((DataNode *)p)->pdirent->d_name,
                  ((DataNode *)q)->pdirent->d_name);
}

```

compare 함수는 파일 이름과 비교하고 list_sort 함수에 사용된다.

```

void do_ls(const char *dir)
{
    DIR *pdir;
    List list;
    create_list(&list, NULL);
    struct dirent *pdirent;
    pdir = opendir(dir);
    if(pdir == NULL){
        perror("opendir failed");
        //exit(1);
    }
    while((pdirent = readdir(pdir)) != NULL){
        if(strcmp(pdirent->d_name, ".") == 0 || strcmp(pdirent->d_name, "..") == 0 )
            continue;
        //show_info(dir, pdirent->d_name);
        DataNode *dn = (DataNode *)malloc(sizeof(DataNode));
        dn->pdirent = pdirent;
        list_push_back(&list, (_Node *)dn);
    }
    //show_info
    //printf("total %d\n", list_size(&list));
    list_sort(&list, compare);
    DataNode *first = (DataNode *)list_begin(&list);
    DataNode *last = (DataNode *)list_end(&list);
    while(first != last){
        show_info(dir, first->pdirent->d_name);
        first = (DataNode *)list_next(&list, (_Node *)first);
        if(first == last){
            printf("\n");
        }
    }
    delete_list(&list);
    closedir(pdir);
}

```

이 함수는 디렉토리의 내용을 출력한다. opendir 로 디렉토리를 열고 readdir 로 디렉토리 엔트리를 읽는다. list_push_back 으로 리스트에 디렉토리 엔트리를 추가하고 list_sort 로 리스트를 정렬한다. 리스트의 각 엔트리에 대해 show_info 로 파일 정보를 출력한다. delete_list 로 리스트를 삭제하고 closedir 로 디렉토리를 닫는다.

(6) pwd

```

ino_t get_inode(const char *path)
{
    DIR *dir;
    struct dirent *pdirent;
    ino_t inode = 0;
    dir = opendir(path);
    while((pdirent = readdir(dir)) != NULL){
        if(strcmp(pdirent->d_name, ".") == 0){
            inode = pdirent->d_ino;
            break;
        }
    }
    closedir(dir);
    return inode;
}

```

get_inode 함수는 주어진 디렉토리 경로에서 해당 디렉토리의 inode 번호를 반환한다.

```

void get_dir_name(ino_t inode, char *buf, int size)
{
    DIR *dir;
    struct dirent *pdirent;
    dir = opendir(".");
    while((pdirent = readdir(dir)) != NULL){
        if(pdirent->d_ino == inode){
            strncpy(buf, pdirent->d_name, size);
            break;
        }
    }
    closedir(dir);
}

```

get_dir_name 함수는 주어진 inode 번호에 해당하는 디렉토리의 이름을 버퍼에 복사한다.

```

void do_pwd()
{
    char dirname[256];
    char path[1024] = "";
    ino_t inode, parent_inode;
    while (1) {
        inode = get_inode(".");
        chdir("..");
        parent_inode = get_inode(".");
        if (inode == parent_inode) {
            break; // Reached root directory
        }
        get_dir_name(inode, dirname, sizeof(dirname));
        // Append the directory name to the path
        char temp[1024];
        snprintf(temp, sizeof(temp), "%s%s", dirname, path);
        strcpy(path, temp);
    }
    // Print the path
    printf("%s", path);
    // Change back to the original directory
    chdir(path + 1); // +1 to skip the initial '/'
}

```

현재 디렉토리 경로를 'path'에 누적하여 저장하고 루트 디렉토리에 도달할 때까지 상위 디렉토리로 이동한다. 최종 경로를 출력후 chdir 명령어로 원래 디렉토리로 돌아간다.

(7) mv

```

in_fp = fopen( srcPath, "r" );
if ( in_fp == NULL )
{
    printf( "fopen()...failed, srcPath=(%s) not found\n", srcPath );
    return -1;
}

```

원본 파일을 읽기 모드로 열고 실패 시 오류 메시지를 출력하고 함수 종료.

```

out_fp = fopen( dstPath, "w" );
if ( out_fp == NULL )
{
    printf( "fopen()...failed, dstPath=(%s) not found\n", dstPath );

    fclose(in_fp);
    return -1;
}

```

타겟 파일을 쓰기 모드로 열고 실패시 오류 메시지 출력 후 원본 파일을 닫은 후 함수 종료

```

while( 0 < (n_size = fread( buff, 1, sizeof(buff), in_fp)) )
{
    fwrite( buff, 1, n_size, out_fp );
}

```

fread 와 fwrite 를 사용하여 원본 파일의 내용을 타겟 파일에 쓴다.

```

fclose( in_fp );
fclose( out_fp );

len += sprintf( szTemp, "File Move Success.\n" );
len += sprintf( &szTemp[len], " - srcPath    = (%s)\n", srcPath );
len += sprintf( &szTemp[len], " - dstPath    = (%s)\n", dstPath );
printf( "%s", szTemp );

```

복사 완료 후 fclose 를 사용하여 파일 포인터를 닫고 이동 성공시 성공을 알리는 메시지를 출력한다.

```

retVal = unlink( srcPath );
if ( retVal == -1 )
{
    printf( "unlink()...failed, retVal=%d\n", retVal );
    return -1;
}

return 0;

```

unlink 를 사용하여 원본 파일을 삭제한다. 실패 시 오류 메시지 출력 후 함수를 종료한다.

Ⅲ. 채팅 프로그램 및 스케줄러

1. IPC

IPC(Inter-Process Communication)는 동일한 시스템에서 실행 중인 여러 프로세스가 서로 데이터를 교환하거나 통신하기 위한 메커니즘을 말한다. IPC는 운영 체제의 중요한 기능 중 하나로, 여러 프로세스가 협력하여 작업을 수행하거나 데이터를 공유할 때 사용된다. IPC 메커니즘은 다양한 방법을 포함하며, 각 방법은 특정한 상황과 요구에 적합하다.

1) Pipes

- 익명 파이프: 한 프로세스에서 데이터를 쓰고 다른 프로세스에서 읽는 단방향 통신 방법이다. 보통 부모 프로세스와 자식 프로세스 간의 통신에 사용된다.
- Named 파이프: FIFO(First In, First Out)로도 알려져 있으며, 이름을 가진 파이프이다. 동일 시스템 내의 서로 다른 프로세스 간에도 통신 할 수 있다.

2) Message Queues

- 메시지를 큐에 저장하고 이를 다른 프로세스가 읽을 수 있도록 하는 방식이다. 메시지는 식별자와 함께 큐에 저장하며, 여러 프로세스가 메시지를 보내고 받을 수 있다.

3) Shared Memory

- 두 개 이상의 프로세스가 동일한 메모리 공간을 공유하여 데이터를 교환하는 방법이다. 매우 빠른 통신 방법이지만, 동기화 문제를 해결하기 위해 추가적 메커니즘이 필요하다.

4) Semaphores

- 주로 프로세스 간의 동기화를 위해 사용된다. 세마포어는 카운터를 사용하여 여러 프로세스가 공통 자원을 접근하는 것을 제어한다.

5) Sockets

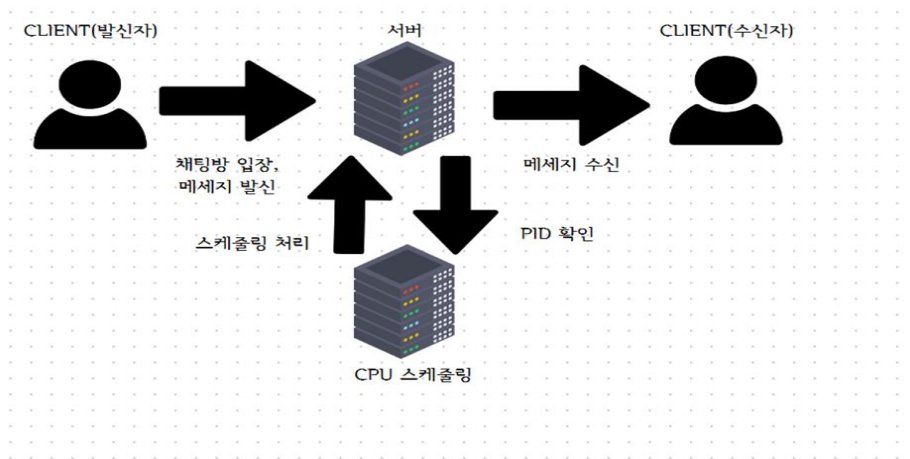
- 네트워크 상의 다른 시스템과 통신하기 위한 인터페이스이다. 로컬 시스템 내의 프로세스 간 통신에도 사용할 수 있다.

6) Signals

- 프로세스에 특정 이벤트나 상태 변화를 알리기 위해 사용되는 소프트웨어 인터럽트이다. 주로 비동기 이벤트 처리를 위해 사용된다.

이러한 수 많은 방식 중, **Socket**을 활용한 통신을 직접 구현하고 성능의 향상을 위해 운영체제 교과목에서 학습한 내용을 활용하자는 주제를 설정하였다.

2. 구조



소켓 프로그래밍으로 Server Client 소켓 프로그래밍은 네트워크 상에서 두 개 이상의 컴퓨터가 서로 데이터를 주고받기 위해 사용하는 통신방법이며, 이 구조에서 서버는 클라이언트의 요청을 듣고 처리하는 역할을 하며, 클라이언트는 서버에 요청을 보내고 응답을 받는 역할을 한다. 이를 활용해 여러 클라이언트가 동시에 서버에 접속하여 메시지를 주고 받을때, 서버는 각 클라이언트의 메시지를 다른 모든 클라이언트에게 브로드캐스트 하는 역할을 한다.

3. Client 동작 과정

```
printf("Configuring remote address...\n");
struct addrinfo hints;
memset(&hints, 0, sizeof(hints));
hints.ai_socktype = SOCK_STREAM;
struct addrinfo *peer_address;
if (getaddrinfo(argv[1], argv[2], &hints, &peer_address)) {
    fprintf(stderr, "getaddrinfo() failed. (%d)\n", GETSOCKETERRNO());
    return 1;
}

printf("Remote address is: ");
char address_buffer[100];
char service_buffer[100];
getnameinfo(peer_address->ai_addr, peer_address->ai_addrlen,
            address_buffer, sizeof(address_buffer),
            service_buffer, sizeof(service_buffer),
            NI_NUMERICHOST);
printf("%s %s\n", address_buffer, service_buffer);

printf("Creating socket...\n");
SOCKET socket_peer;
socket_peer = socket(peer_address->ai_family,
                    peer_address->ai_socktype, peer_address->ai_protocol);
if (!ISVALIDSOCKET(socket_peer)) {
    fprintf(stderr, "socket() failed. (%d)\n", GETSOCKETERRNO());
    return 1;
}
```

get_addr_info 를 사용하여 원격 서버 주소와 포트를 설정하고, get_name_info 로 해당 주소를 출력하며, socket 함수를 통해 TCP 소켓을 생성하여 서버와 통신을 준비하는 형태로 Client 측의 구현을 완료하였다.

```
while(1) {

    fd_set reads;
    FD_ZERO(&reads);
    FD_SET(socket_peer, &reads);
    FD_SET(0, &reads);

    struct timeval timeout;
    timeout.tv_sec = 0;
    timeout.tv_usec = 100000;

    if (select(socket_peer+1, &reads, 0, 0, &timeout) < 0) {
        fprintf(stderr, "select() failed. (%d)\n", GETSOCKETERRNO());
        return 1;
    }

    if (FD_ISSET(socket_peer, &reads)) {
        char read[4096];

        int bytes_received = recv(socket_peer, read, 4096, 0);
        if (bytes_received < 1) {
            printf("Connection closed by peer.\n");
            break;
        }
        printf("Received (%d bytes): %.*s",
            bytes_received, bytes_received, read);
    }

    if(FD_ISSET(0, &reads)) {
        char read[4096];
        if (!fgets(read, 4096, stdin)) break;
        printf("Sending: %s", read);
        int bytes_sent = send(socket_peer, read, strlen(read), 0);
        printf("Sent %d bytes.\n", bytes_sent);
    }
} //end while(1)
```

코드는 'select' 함수를 사용하여 소켓과 표준 입력을 동시에 모니터링하여, 서버로부터 수신된 메시지를 출력하고 사용자 입력을 서버로 전송하는 무한 루프 형태로 구현하였다. 한개의 프로세스를 통해 많은 통신들을 관리한다.

4. Server 동작 과정

TCP 서버를 설정하고 클라이언트 연결을 처리하며, 여러 클라이언트와의 통신을 스레드를 사용하여 병렬로 처리하는 구조로 설계하였다.


```

void *handle_client(void *arg) {
    SOCKET client_socket = *(SOCKET *)arg;
    free(arg);
    char read[1024];
    while (1) {
        int bytes_received = recv(client_socket, read, 1024, 0);
        if (bytes_received < 1) {
            printf("완료\n");
            FD_CLR(client_socket, &master);
            CLOSESOCKET(client_socket);

            return NULL;
        }
        // sleep(8);

        for (SOCKET j = 0; j <= max_socket; ++j) {

            if (FD_ISSET(j, &master)) {
                if (j != client_socket && j != socket_listen) {
                    printf("전송완료\n");
                    pthread_mutex_lock(&socket_mutex);
                    send(j, read, bytes_received, 0);
                    pthread_mutex_unlock(&socket_mutex);
                }
            }
        }
    }

    return NULL;
}

```

클라이언트와의 통신을 처리하는 스레드 함수이고, 클라이언트로부터 메시지를 수신하고, 다른 모든 클라이언트에게 브로드캐스트 한다.

```

printf("Configuring local address...\n");
struct addrinfo hints;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;

struct addrinfo *bind_address;
getaddrinfo(0, "8080", &hints, &bind_address);

printf("Creating socket...\n");
socket_listen = socket(bind_address->ai_family, bind_address->ai_socktype, bind_address->ai_protocol);
if (!ISVALIDSOCKET(socket_listen)) {
    fprintf(stderr, "socket() failed. (%d)\n", GETSOCKETERRNO());
    return 1;
}

```

로컬 주소를 설정하고, TCP 소켓을 생성한다. 주소 정보는 'getaddrinfo' 함수를 통해 가져온다.

```

printf("Binding socket to local address...\n");
if (bind(socket_listen, bind_address->ai_addr, bind_address->ai_addrlen)) {
    fprintf(stderr, "bind() failed. (%d)\n", GETSOCKETERRNO());
    return 1;
}
freeaddrinfo(bind_address);

printf("Listening...\n");
if (listen(socket_listen, 10) < 0) {
    fprintf(stderr, "listen() failed. (%d)\n", GETSOCKETERRNO());
    return 1;
}

```

소켓을 로컬 주소에 바인딩하고, 클라이언트의 연결 요청을 듣기 위해 Listening 상태로 설정한다.

```

while (1) {
    fd_set reads;
    reads = master;
    if (select(max_socket + 1, &reads, 0, 0, 0) < 0) {
        fprintf(stderr, "select() failed. (%d)\n", GETSOCKETERRNO());
        return 1;
    }

    for (SOCKET i = 0; i <= max_socket; ++i) {
        if (FD_ISSET(i, &reads)) {
            if (i == socket_listen) {
                struct sockaddr_storage client_address;
                socklen_t client_len = sizeof(client_address);
                SOCKET socket_client = accept(socket_listen, (struct sockaddr *)&client_address, &client_len);
                if (!ISVALIDSOCKET(socket_client)) {
                    fprintf(stderr, "accept() failed. (%d)\n", GETSOCKETERRNO());
                    return 1;
                }

                FD_SET(socket_client, &master);
                if (socket_client > max_socket)
                    max_socket = socket_client;

                char address_buffer[100];
                getnameinfo((struct sockaddr *)&client_address, client_len, address_buffer, sizeof(address_buffer), 0, 0, NI_NUMERICHOST);
                printf("New connection from %s\n", address_buffer);

                SOCKET *pclient = malloc(sizeof(SOCKET));
                *pclient = socket_client;
                pthread_t thread_id;

                pthread_create(&thread_id, NULL, handle_client, pclient);
                printf("스레드 만들기");
                pthread_detach(thread_id);
            }
        }
    }
}

```

'FD_ZERO', 'FD_SET'을 사용하여 파일 디스크립터 셋을 초기화하고, Listening 소켓을 추가했다. 이는 새로운 클라이언트 연결을 수락하고, 각 클라이언트를 별도의 스레드로 처리한다.

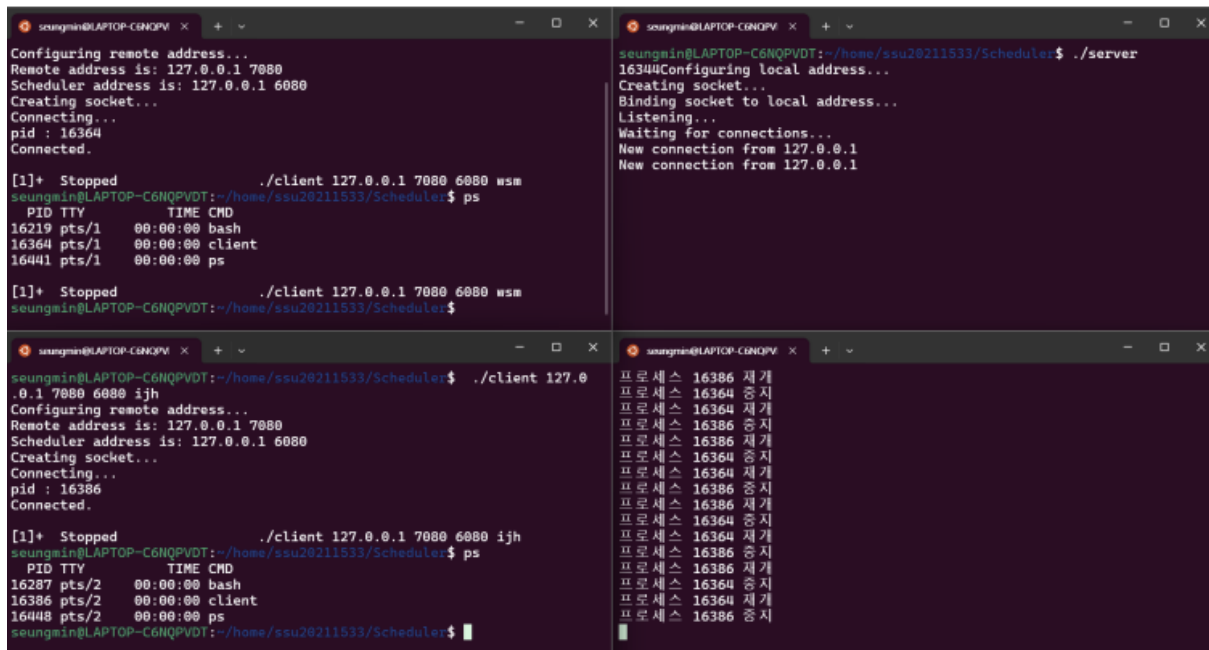
III-1. Scheduling

1. Round Robin

라운드 로빈 스케줄링은 각 프로세서가 공정하게 CPU 시간을 할당받아 실행 되도록한다. 각 프로세스는 정해진 시간 할당 동안 CPU 를 사용한다. 한 프로세스가 시간 할당을 모두 사용하면 준비 큐의 맨 뒤로 이동하고, 다음 프로세스가 CPU 를 할당 받는다.

2. 사용이유

채팅 프로그램에 전송될 메시지를 담는 버퍼를 한 클라이언트가 점유하는 것을 방지하기 위해 라운드 로빈 스케줄러를 통해 각 클라이언트에게 시간할당을 해 버퍼를 점유하게 하려 했으나, 문제를 만났다.



```
seungmin@LAPTOP-C6NQPV: ~$ ./server
Configuring remote address...
Remote address is: 127.0.0.1 7080
Scheduler address is: 127.0.0.1 6080
Creating socket...
Connecting...
pid : 16364
Connected.

[1]+  Stopped                  ./client 127.0.0.1 7080 6080 wsm
seungmin@LAPTOP-C6NQPV: ~/home/ssu20211533/Scheduler$ ps
  PID TTY          TIME CMD
 16219 pts/1    00:00:00 bash
 16364 pts/1    00:00:00 client
 16441 pts/1    00:00:00 ps

[1]+  Stopped                  ./client 127.0.0.1 7080 6080 wsm
seungmin@LAPTOP-C6NQPV: ~/home/ssu20211533/Scheduler$

seungmin@LAPTOP-C6NQPV: ~/home/ssu20211533/Scheduler$ ./client 127.0.0.1 7080 6080 ijh
Configuring remote address...
Remote address is: 127.0.0.1 7080
Scheduler address is: 127.0.0.1 6080
Creating socket...
Connecting...
pid : 16386
Connected.

[1]+  Stopped                  ./client 127.0.0.1 7080 6080 ijh
seungmin@LAPTOP-C6NQPV: ~/home/ssu20211533/Scheduler$ ps
  PID TTY          TIME CMD
 16287 pts/2    00:00:00 bash
 16386 pts/2    00:00:00 client
 16448 pts/2    00:00:00 ps
seungmin@LAPTOP-C6NQPV: ~/home/ssu20211533/Scheduler$

seungmin@LAPTOP-C6NQPV: ~/home/ssu20211533/Scheduler$ ./server
16344Configuring local address...
Creating socket...
Binding socket to local address...
Listening...
Waiting for connections...
New connection from 127.0.0.1
New connection from 127.0.0.1

프로세스 16386 재개
프로세스 16364 중지
프로세스 16364 재개
프로세스 16386 중지
프로세스 16386 재개
프로세스 16364 중지
프로세스 16364 재개
프로세스 16386 중지
프로세스 16386 재개
프로세스 16364 중지
프로세스 16364 재개
프로세스 16386 중지
프로세스 16386 재개
프로세스 16364 중지
프로세스 16364 재개
프로세스 16386 중지
```

이와 같이, 현재 구현한 CPU 스케줄러가 프로세스를 중지 시켰을 때 프로세스는 여전히 유지되며 BASH 가 종료되는 현상을 만났다. 이후 while 문으로 두 프로세스의 숫자를 증가 시켰을 때 SIGSTOP 과 SIGCONT 모두 동작 하는 것은 확인했으나 다시 초기 화면인 채팅화면으로는 돌아가지 않는다는 문제를 발견했다. (따라서 SIGSTOP 으로 구현 시 채팅은 불가하다.)

III-2. Multi Threading

1. 이론적 배경

1) Process

프로세스는 실행 중인 프로그램의 인스턴스를 의미한다. 즉, 메모리에 로드되어 CPU 에 의해 실행되는 프로그램의 상태이다. 각 프로세스는 독립적인 메모리 공간을 가지며, 운영 체제에 의해 자원을 할당받아 실행된다.

2) Thread

쓰레드는 프로세스 내에서 실행되는 흐름의 단위이다. 쓰레드는 프로세스 내의 코드, 데이터 및 시스템 리소스에 대한 공유를 허용하므로, 여러 작업을 병렬로 처리하거나 동시에 여러 작업을 수행할 수 있다. 주로 프로그램의 성능 향상을 위해 멀티 쓰레드 구조로 프로그램을 설계한다.

3. Multi Thread

멀티 쓰레드는 하나의 프로세스 내에서 여러 개의 쓰레드가 동시에 실행되는 것을 의미한다. 여러 작업을 병렬로 처리하여 전체적인 작업 처리 속도를 향상시킬 수 있다. 하지만, 쓰레드의 분기로 인해 발생하는 자원 공유 문제를 추가로 고려해야 한다.

2. 적용 배경과 실제 적용

1) 적용 배경

SERVER의 소켓을 받아서, 해당 소켓을 CLIENT에게 전송하는 과정에서 기본적으로는 싱글 프로세스 형태로 구현하였다. 이러한 구조에 맞추어, CPU 스케줄링을 활용하여 SERVER에서 소켓을 관리하는 형태로 최초 고안하였으나 해당 방식의 적용 이후 프로세스의 성능을 높임과 동시에 소켓에서 발생할 수 있는 동시성 문제를 해결하자는 의견이 제시되었다. 이에 따라, Multi Threading 형태로 변경하고 이후 동기화를 통해 소켓에 대한 동시성 문제를 해결하였다.

2) 실제 적용

```
pthread_t thread_id;

pthread_create(&thread_id, NULL, handle_client, pclient);
printf("스레드 분기");
pthread_detach(thread_id);
```

server 측의 코드로써, connection이 진행된 이후 새로운 쓰레드를 생성하고, 해당 쓰레드에서 handle_client를 진행하는 형태로 변경하였다.

```
void *handle_client(void *arg) {
    SOCKET client_socket = *(SOCKET *)arg;
    free(arg);
    char read[1024];
    while (1) {
        int bytes_received = recv(client_socket, read, 1024, 0);
        if (bytes_received < 1) {
            printf("완료\n");
            FD_CLR(client_socket, &master);
            CLOSESOCKET(client_socket);
            return NULL;
        }

        for (SOCKET j = 0; j <= max_socket; ++j) {
            if (FD_ISSET(j, &master)) {
                if (j != client_socket && j != socket_listen) {
                    printf("전송완료\n");
                    send(j, read, bytes_received, 0);
                }
            }
        }
    }
}
```

handle client 함수의 구현은 다음과 같으며, 메시지를 받아와 이를 send 하는 부를 함수로 분리하였다. 일련의 과정을 통해 멀티 쓰레드 환경에서 동작하는 서버를 구현하였다.

3) 성능 확인

```
New connection from 127.0.0.1  
전송완료  
Message sent at: 1717839769 seconds, 517055 microseconds
```

싱글 스레드 환경에서 동작하는 서버에 대해 소요 시간을 확인한 것. 소요 시간을 측정한 결과 510755 microsecond 가 나왔다.

```
전송완료  
Message sent at: 1717839625 seconds, 321515 microseconds
```

멀티 스레드 환경에서 동작하는 서버에 대해 소요 시간을 확인한 것. 소요 시간을 측정한 결과 321515 microsecond 가 나왔다.

이와 같은 형태로, 멀티 스레드를 통해 성능이 향상됨을 확인하였다.

III-3. 동기화

1. 동기화

1) 동기화의 개념

동기화는 병렬 또는 동시에 실행되는 여러 프로세스나 스레드 간에 데이터 접근과 작업을 조정하는 개념이다. 공유된 리소스에 대한 동시 접근을 관리하고, 데이터 일관성을 유지하기 위해 필요하다. 여러 스레드 또는 프로세스가 공유 데이터에 동시에 접근할 때 문제가 발생할 수 있다. 이를 해결하기 위해 동기화 기법이 사용된다.

2) 고안한 이유

서버 측에서 성능을 높이기 위해서, 멀티 스레드로 이전 과정에서 분기를 진행하였다. 이러한 분기에 따라 각 스레드 별로 **SOCKET** 을 공유 자원으로 활용하게 되었다. 이로 인해 해결하고자 하였던 문제들은 다음과 같다.

3) 발생할 수 있는 문제들

- 소켓의 메시지를 전송 중, 메시지가 송신 되었을 때의 문제
- 수신 도중, 메시지를 송신함으로써 발생할 수 있는 문제.

2. 구현한 최종 코드

```

void *handle_client(void *arg) {
    SOCKET client_socket = *(SOCKET *)arg;
    free(arg);
    char read[1024];

    while (1) {
        if (pthread_mutex_lock(&socket_mutex) != 0) {
            printf("lock을 얻지 못함\n");
            return NULL;
        }
        printf("lock을 얻음\n");
        int bytes_received = recv(client_socket, read, 1024, 0);
        if (bytes_received < 1) {
            pthread_mutex_unlock(&socket_mutex);
            FD_CLR(client_socket, &master);
            CLOSESOCKET(client_socket);
            return NULL;
        }

        for (SOCKET j = 0; j <= max_socket; ++j) {
            if (FD_ISSET(j, &master)) {
                if (j != client_socket && j != socket_listen) {
                    send(j, read, bytes_received, 0);
                }
            }
        }

        if (pthread_mutex_unlock(&socket_mutex) != 0) {
            printf("lock을 반납 못함\n");
            return NULL;
        }
    }
}

```

위 코드는 서버가 클라이언트에게 데이터를 받았을 때 처리하는 코드이다.

데이터를 받은 상태에서 lock 을 얻고 함수가 끝나는 시점에서 lock 을 반납하는 방식이다.

하지만 이 방식은 클라이언트가 메시지를 보낸 뒤 반납하지 않고, 스레드가 종료될 때 lock 을 반납하게 되었다. 이유는 스레드 생성시에 함수를 매개변수로 넣어서 생성하기 때문에 해당함수를 종료하는 시점이 스레드가 종료되는 시점이었기 때문이다.

이 문제를 해결하기 위해서 다음과 같이 코드를 수정했다.

```

void *handle_client(void *arg) {
    SOCKET client_socket = *(SOCKET *)arg;
    free(arg);
    char read[1024];
    while (1) {
        // 데이터 수신
        int bytes_received = recv(client_socket, read, 1024, 0);
        if (bytes_received < 1) {
            printf("완료\n");
            pthread_mutex_lock(&socket_mutex);
            FD_CLR(client_socket, &master);
            pthread_mutex_unlock(&socket_mutex);
            CLOSESOCKET(client_socket);
            return NULL;
        }

        // 데이터 전송
        for (SOCKET j = 0; j <= max_socket; ++j) {
            if (FD_ISSET(j, &master)) {
                if (j != client_socket && j != socket_listen) {
                    if (pthread_mutex_trylock(&socket_mutex) == 0) {
                        sleep(6);
                        printf("전송완료\n");
                        send(j, read, bytes_received, 0);
                        pthread_mutex_unlock(&socket_mutex);
                    } else {
                        printf("뮤텍스가 이미 잠겨 있습니다.\n");
                    }
                }
            }
        }
    }

    return NULL;
}

```

해당 코드는 lock의 적용 시점을 서버가 클라이언트에게 메시지를 send()하는 시점으로 변경해서 문제를 해결했다.

IV. 결론

이번 프로젝트에서 우리는 운영체제의 중요한 개념들을 직접 구현하고 실험해보는 과정을 통해 많은 것을 배웠다. IPC(Inter-Process Communication) 메커니즘을 이용한 채팅 프로그램, 다양한 터미널 명령어 구현, 그리고 스케줄링 및 멀티 스레딩 기술을 적용하면서 이론적인 지식을 실제 코드로 구현할 수 있는 능력을 키울 수 있었다. 이 프로젝트를 통해 우리는 운영체제의 이론적 지식을 실제 구현해보는 값진 경험을 쌓았으며, 이를 통해 향후 더 복잡한 시스템을 설계하고 구현하는 데 필요한 기초를 다질 수 있었다. 앞으로도 이러한 경험을 바탕으로 더 많은 도전을 통해 성장할 수 있기를 기대한다.

추후에 스레드 동기화처리 부분에서 스레드의 종료가 아닌 메시지의 전송완료 시점으로 하여 여러 클라이언트가 통신을 할 수 있으므로 이 부분을 보완하고, 파일 명령어 구현시 옵션을 구현하지 못한 것이 아쉬웠다. 또, IPC 에서 프로세스들이 통신할 때 다른 주소에서도 서버에 접속해 채팅을 하는 포트 포워딩을 더 공부해 구현하고 싶다.

V. 참고문헌

- Evaluation of Inter-Process Communication Mechanisms, Aditya Venkataraman, Kishore Kumar Jagadeesha 2023
- TCP/IP Sockets in C: Practical Guide for Programmers, Michael J. Donahoo, Kenneth L. Calvert
- An Optimized Round Robin Scheduling Algorithm for CPU Scheduling, Ajit Singh, Priyanka Goyal, Sahil Batra
- Evaluation of Inter-Process Communication Mechanisms, Aditya Venkataraman, Kishore Kumar Jagadeesha 2015